A property may have three different values:

|  |  |
| --- | --- |
| Value type | Description |
| Pre-conversion value | The value before it is converted to the type of the property. It is usually set by the View and may be of a different type than the property. |
| Pre-validation value | The value after it is converted to the correct type but before the strongly typed validation is performed. |
| Post-validation value | The value after all validations has succeeded. This is usually the same value as in the source objects. |

## Validation

**Validation** means the process of validating a property or view model. A **validator** in contrast means the method, object or class that is capable of doing the validation.

There are two types of validations:

**Property validations** are performed when the value of a property is about to change. The validation checks the new value the property is going to assume. If all property validations of a property succeed, the new value is written to the source object. Otherwise the new value is cached as the pre-validation value of the property, but the source object remains unchanged.

**View model validations** are performed every time the view model it targets or any of its descanting view models has changed (a property has changed or the validation state has changed). It is important to note that view model validations are performed after a property has changed which means they do not prevent the modification of the source objects.

The **validation state** is the result of a validation. It holds all validation errors and the overall result (valid/invalid). Each property and the view model have a validation state.

### Validation scope

The validation scope specifies what part of the VM hierarchy is revalidated.

|  |  |
| --- | --- |
| Validation scope | Description |
| Self only | Only the current VM itself is validated. All validations that ancestors defined for this VM are also performed. |
| Full sub tree | The current VM and all its descendants are revalidated. |
| Self and validated children | The current VM is revalidated and all children, for which the current VM has defined validations. |

### Validation mode

The validation mode defines how revalidation is performed.

|  |  |
| --- | --- |
| Validation mode | Description |
| Commit valid value | A property may have invalid pre-conversion and pre-validation values. If you revalidate with this option the VM revalidates the invalid values write them to the source object it the have become valid. |
| Discard invalid value | The pre-conversion and pre-validation values of the property are discarded and the post-validation is revalidated. |

### The validation process

#### Revalidate view model

**Main success scenario** (validation mode self only):

1. The VM revalidates all properties with the specified validation mode.
2. The VM performs view model validations.

**Extensions:**

1a. The revalidation scope was full sub tree:

1a1. Before validating the properties, revalidate the children of the VM with validation scope full sub tree and the given validation mode.

1b. The validation scope was self and validated children:

1b1. Before validating the properties, revalidate all children for which the current VM has validations defined.

#### Revalidate property (commit valid values)

**Main success scenario:**

1. The VM gets the pre-conversion value of the property.
2. The VM sets the pre-conversion value of the property, which performs the property validation.

#### Revalidate property (discard invalid values)

**Main success scenario:**

1. The VM clears the pre-conversion and pre-validation cache.
2. The VM performs a property validation on the pre-validation value of the property.

#### Perform view model validations

**Main success scenario:**

1. The VM raises the view model validating event.
2. The VM forwards the event to all ancestor VMs.
3. The VM or any ancestor VM may handle the event and add validation errors to the validation state.
4. The VM saves the new validation state.
5. The VM compares the new validation state to the previous validation state and raises a validation state changed event if it has changed.

#### Perform property validation

**Main success scenario:**

1. The VM raises the property validating event.
2. The VM forwards the event to all ancestor VMs.
3. The VM or any ancestor VM may handle the event and add validation errors to the validation state.
4. The VM saves the new validation state.
5. The VM compares the new validation state to the previous validation state and raises a validation state changed event if it has changed.

Validate on property change

Main success scenario:

### Validation context

The **validation context** holds context that is shared during a single validation process. It holds various validator contexts and the revalidation queue.

### Revalidation queue

If a validator validates other VMs in addition to its target VM and notices that their validation state may have changed, it can add them to the **revalidation queue**. All VMs in the revalidation queue are validated sometime after the current validator invocation has finished.

### Validator context

If a parent VM type P defines a validator for an ancestor VM type A that validator may be called many times during a single validation process when following conditions apply:

1. There is a collection relation anywhere between P and A.
2. Either a full validation is performed (meaning that every item of the collection is validated) or a validator defined for A adds other instances of A to the revalidation queue.

In this case the same **validator context** is passed to all invocations of the validator that are performed for a certain instance of P. Note that validator invocations of two different instances of P do not share the same validator context.

### Validation arguments

|  |  |
| --- | --- |
| Value type | Description |
| Owner VM | The VM whose descriptor has defined the validator. |
| Target VM | The VM that should be validated. |
| Target path | A VM instance path that contains all VMs from the owner VM to the target VM (including both). |
| Target property | The property that should be validated. This property is only valid for property validations. |
| Changed VM | The VM whose change has triggered the validation. In case of a property definition it contains the VM that is about to change. |
| Changed path | A VM instance path that contains all VMs from the target VM to the changed VM (including both). |
| Changed property | The property that has changed (if any). This property is only valid for view model validations. |
| Errors | Holds all validation errors that occur in the current validation process. |

Example 1: A validator defined on the createViews VM is called because a property of it has changed.



Example 2: A validator defined on the list VM is called because the createViews VM has changed.



Old documentation

1. Execute all view model validations that are defined for the current VM.
2. If the validation state has changed raise a validation changed event.

### 

### Collection validation

Validation process for leanPlan:

1. Validate all children of leanPlan. The only child is the Tasks collection. Validate collections like follows:
   1. Create a new validation state holder.
   2. Call validate for each TaskVM (createDal, developSetup) with the state holder.
   3. For the TaskVM do the following:
      1. Validate the children of createDal (none in this case).
      2. Execute the property validations of createDal:
         1. Call PropertyValidating of the IPropertyBehaviorContext. All VMs up the hierarchy have the opportunity to handle the validation event and add errors.
         2. If the validation state has changed (valid/invalid, collection of errors), raise a notification.
      3. Execute the view model validations of createDal.
         1. Call ViewModelValidating of the IViewModelBehaviorContext. All the VMs up the hierarchy have the opportunity to handle the validation event and add errors.
         2. If the validation state has changed, raise a notification.
      4. If the previous steps raised a validation state changed notification, all ancestor VMs handle it. Do the following in their event handler:
         1. Start a view model validation process.
         2. Execute all view model validations for the VM.
         3. If the validation state has changed, raise a notification (which again bubbles up).
2. Execute the property validations of leanPlan.
3. Execute the view model validations of leanPlan.

#### Using the custom state to optimize collection validators for full validations of collections

Some collection validators may need to consider all collection items to determine if a single item is valid. A prominent example is the unique validation. Consider the following steps:

1. Validate the Tasks collection:
   1. Execute the unique validation of Tasks for each TaskVM:
      1. Search all items of Tasks if there is another task that has the same name as the current but that is not the current task. If there is such a task, add a validation error.

But for large collections this may result in a bad performance because for every item in the list we have to iterate over all other items, which results in n² operations.

To optimize this, a validator may get or set a custom state object. The state object set by a validator is passed to every validate operation that is called for the remaining items of collection. With this technique we implement an optimized version of the unique validation:

1. Validate the task collection:
   1. Execute the unique validation of each TaskVM:
      1. If the custom state object is null (this is case for the first item), create

## Behaviors

### Motivation

View models often have to implement several cross-cutting concerns such as validation, synchronization with the domain model, change notification, authorization and inter-property dependencies.

Implementing all this concerns in one class is not very practical. This is further complicated because there are many different combinations of these concerns varying between VM properties and between view models. The result is often lots of code repetition and code that is complex and hard to maintain.

To overcome these problems most of the functionality in this framework is implemented as behaviors.

### Behaviors

View models and VM properties provide several operations on them such as GetValue, GetValidationState, Revalidate, UpdateFromSource, etc.

These operations are not implemented by the view models and properties themselves but are delegated to a behavior chain. Each operation is defined in a behavior interface (the terms object, class and interface refer to the .NET concepts in this section). A behavior chain is a linked list of behavior objects. Each behavior object (or more exactly its class) may implement one or more behavior interfaces. Each VM property and the view model have their own behavior chain.

When an operation is for a view model or VM property is invoked, the first behavior object in the behavior chain (of the view model or VM property) that implements the behavior interface on which the requested operation is defined is invoked.

A behavior interface method implementation may (and usually should) call the next behavior in the chain that also implements the behavior interface (if there is such an object). It can execute arbitrary code before or after it forwards the operation to the next behavior. It may also call operations on other behavior interfaces implemented by behavior objects that come anywhere after the current object in the chain.

This means that a single operation can be processed by multiple behavior objects/classes. This pattern is based on the Chain-of-responsibility design pattern.

[TBD: graphical illustrations, examples]

### Behavior configuration

Different VMs and VM properties may have different behaviors. These behaviors are determined by the VMDescriptionBuilder (methods like WithProperties, WithValidations, WithBehaviors may all add or configure behaviors).

What behaviors a chain may have and the order of them is determined by a BehaviorChainTemplate. Because there are main BehaviorChainTemplates, all available templates are managed the BehaviorChainTemplateRegistry. A template is identified by a template key (an arbitrary object).

The VMDescriptorBuilder creates BehaviorChainConfigurations based on the templates which are used to hold transient behavior settings while the VM descriptor is constructed. The BehaviorChainConfigurations of the VM and all VM properties are managed by a VMDescriptorConfiguration object.

At the end the VMDescriptorBuilder creates the concrete behavior chains from the BehaviorChainConfigurations.